Python for Data Science & AI

Dasun Athukoralage

web: www.dasuna.me
email: dasun@nirvanaclouds.com

Built in functions

---

sum():

The built-in sum() function in Python is used to **add up all the elements** in an iterable (like a list, tuple, or set).

Built in functions

---

sum():

Syntax:

sum(iterable, start)

- **‘iterable’:** A sequence (e.g., list, tuple) containing numbers.
- **‘start’(optional):** A number that is added to the total. Default is 0.

-

Built in functions

---

sum():

Example 1:

```
numbers = [1, 2, 3, 4, 5]

total = sum(numbers)

print(total)  # Output: 15
```

Built in functions

---

sum():

Example 2:

```
numbers = [1, 2, 3, 4, 5]

total = sum(numbers, 10)

print(total)  # Output: 25 (1+2+3+4+5+10)
```

Built in functions

---

isinstance():

The built-in isinstance() function in Python is used to **check the type** of an object.

Built in functions

---

isinstance():

Syntax:

```
isinstance(object, classinfo)
```

- **‘object’:** The variable or value you want to check.
- **‘classinfo’:**The type (or a tuple of types) you want to check against.

-

Built in functions

---

isinstance():

Example 1:

```
x = 10

print(isinstance(x, int))  # Output: True
```

Built in functions

---

isinstance():

Example 2:

```
x = 3.14
print(isinstance(x, (int,float)))  # Output: True
```

What are f-strings?

---

f-strings are a way to embed expressions inside string literals using curly braces {}.

They were introduced in **Python 3.6**.

What are f-strings?

The Core Syntax: `f` and `{}`

The magic of f-strings comes from two simple parts:

- **The `f` prefix:** A string becomes an f-string simply by placing an `f` or `F` before the opening quote.
- **The curly braces `{}`:** Any valid Python code you place inside curly braces within the f-string will be evaluated, and its result will be inserted into the string at that position.

What are f-strings?

Example 1:

```
name = "Alice"
age = 30
print(f"My name is {name} and I am {age} years old.")

#Output

My name is Alice and I am 30 years old.
```

Modules & Packages

What are ‘Modules’ in Python?

In Python, a "module" refers to **a file that contains Python code**. This code can include variables, functions, and classes that can be used in other Python programs.

Modules are a fundamental concept in Python that allows you to organize your code into **reusable and maintainable components**.

Creating a Module

To create a module, you typically **create a separate .py file** that contains Python code. This file can **include functions, variables, and classes**.

For example, if you create a file named my_module.py with some functions, **you can import those functions in other Python scripts**.

Creating a Module

Let us create a module. Type the following and save it as **my_module.py**.

```
# Python Module addition

def add(a, b):

    result = a + b

    return result
```

Import modules in Python

To use the contents of a module in another Python script, you use the import statement. For example:

```
import my_module
```

This does not import the names of the functions defined in example directly.

It only imports the module name example there.

Import modules in Python

Using the module name we can access the function using the dot (.) operator. For example:

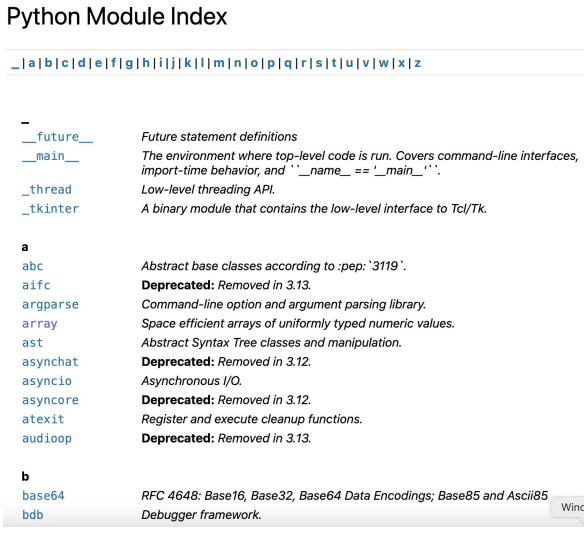
```
my_module.add(4,5)
```

Import Python Standard library modules

The **Python standard library** contains well **over 200 modules**.

We can import a module according to our needs.

https://docs.python.org/3/py-modindex.html



Python Module Index

_ | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | z

_

__future__	*Future statement definitions*
__main__	*The environment where top-level code is run. Covers command-line interfaces, import-time behavior, and ``__name__ == '__main__'``.*
_thread	*Low-level threading API.*
_tkinter	*A binary module that contains the low-level interface to Tcl/Tk.*

a

abc	*Abstract base classes according to :pep:`3119`.*
aifc	**Deprecated:** *Removed in 3.13.*
argparse	*Command-line option and argument parsing library.*
array	*Space efficient arrays of uniformly typed numeric values.*
ast	*Abstract Syntax Tree classes and manipulation.*
asynchat	**Deprecated:** *Removed in 3.12.*
asyncio	*Asynchronous I/O.*
asyncore	**Deprecated:** *Removed in 3.12.*
atexit	*Register and execute cleanup functions.*
audioop	**Deprecated:** *Removed in 3.13.*

b

base64	*RFC 4648: Base16, Base32, Base64 Data Encodings; Base85 and Ascii85*
bdb	*Debugger framework.*

Import Python Standard library modules

Suppose we want to get the **value of pi**, first we **import the math module** and use **math.pi**. For example,

```
# import standard math module

import math

# use math.pi to get value of pi

print("The value of pi is", math.pi)
```

Python import with Renaming

In Python, we can also import a module by renaming it. For example,

```
# import module by renaming it
import math as m

print(m.pi)

# Output: 3.141592653589793
```

Python from ... import statement

We can import specific names from a module without importing the module as a whole. For example,

```
# import only pi from math module
from math import pi

print(pi)

# Output: 3.141592653589793
```

import all names

In Python, we can import all names(definitions) from a module using the following construct.

```
# import all names from the standard module math
from math import *

print("The value of pi is", pi)
```

import all names

Here, we have **imported all the definitions** from the math module. This includes all names visible in our scope except those beginning with an underscore(private definitions).

import all names

Importing everything with the asterisk (*) symbol is **not a good programming practice.** This can lead to duplicate definitions for an identifier. It also hampers the readability of our code.

if __name__ = “__main__” construct

This tells Python to **only run some code when the file is being run directly**, and **not when it’s being imported** as a module in another file.

This construct allows you to write code that **can be both reusable as a module and executable as a standalone program**.

if _ _name_ _ = “_ _main_ _” construct

How it works?

1.) When the script is run directly

When you run a Python script directly from the command line or as the main program, the **Python interpreter sets** a special **built-in variable called __name__ to "__main__"**. This indicates that the script is being executed as the main program.

if _ _name_ _ = “_ _main_ _” construct

How it works?

2.) When the script is imported as a module

When you import a Python script as a module into another script, the **__name__ variable is set to the name of the module** (i.e., the name of the script file) rather than "__main__".

if _ _name_ _ = “_ _main_ _” construct

Example: my_module.py

```
# This code will always run, whether the script is run directly or imported as a module.

def some_function():

    print("This function can be used when the script is imported as a module.")

# The following code will only run if the script is executed directly, not when imported as
a module.

if __name__ == "__main__":

    print("This code runs when the script is executed directly.")

    some_function()
```

if _ _name_ _ = “_ _main_ _” construct

Example:

When you run this script directly, you will see both the "This code runs when the script is executed directly." message and the output of some_function().

However, if you import this script as a module into another Python script, the code inside the **if __name__ == "__main__":** block **will not be executed**, and only the some_function() can be accessed.

Python Packages

What is a Python package?

Package in Python is **a folder that contains various modules as files**.

Analogy	Python Term	Description
A single tool (like a wrench)	**Module**	A single `.py` file with Python code.
A toolbox	**Package**	A folder containing multiple modules.

What is a Python package?

Packages allow for a **hierarchical structuring** of the module namespace using dot notation.

In the same way that **modules help avoid collisions between global variable names**, **packages help avoid collisions between module names**.

Creating a package

Let's create a package in Python named **my_calculator** that will contain two modules **addition** and **subtraction**. To create this package follow the below steps:

- Create a folder named **my_calculator**.
- Inside this folder create an empty Python file i.e. **__init__.py**
- Then create two modules **addition** and **subtraction** in this folder.

Creating a package

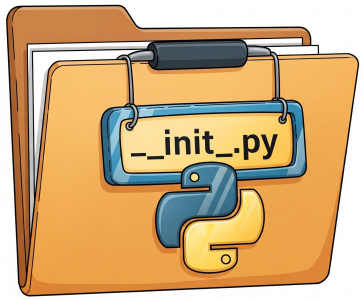
Now our file hierarchy will be as followed.

```
my_calculator/
    __init__.py
    addition.py
    subtraction.py
```

Creating a package

Why __init__.py file?

Think of the **__init__.py** file as a special sign you put on a folder. It tells Python, "Hey, this isn't just a regular folder. Treat it as a single, organized toolbox of code—a **package**.



Creating a package

---

addition.py

```
def add(a, b):
    return a + b
```

Creating a package

---

subtraction.py

```
def subtract(a, b):
    return a - b
```

How to invoke modules and their functions?

To do that, let's create another Python file named '**main.py**' outside the package directory. Now, our file hierarchy will be as follows.

How to invoke modules and their functions ~ 1st way

main.py

```
# main.py
# Import the entire modules
import my_calculator.addition
import my_calculator.subtraction

# Perform some calculations
result1 = my_calculator.addition.add(5, 3)
result2 = my_calculator.subtraction.subtract(10, 4)

# Display the results
print("Addition result:", result1)
print("Subtraction result:", result2)
```

How to invoke modules and their functions ~ 1st way

Pros & Cons

- This form of import, imports the ‘**addition**’ module, but you need to prefix the module name when using its contents.

How to invoke modules and their functions ~ 2nd way

main.py

```
# main.py
# Import the entire modules
from my_calculator import addition
from my_calculator import subtraction

# Perform some calculations
result1 = addition.add(5, 3)
result2 = subtraction.subtract(10, 4)

# Display the results
print("Addition result:", result1)
print("Subtraction result:", result2)
```

How to invoke modules and their functions ~ 3rd way

main.py

```
# main.py
# Import the entire modules
from my_calculator.addition import add
from my_calculator.subtraction import subtract

# Perform some calculations
result1 = add(5, 3)
result2 = subtract(10, 4)

# Display the results
print("Addition result:", result1)
print("Subtraction result:", result2)
```

How to invoke modules and their functions ~ 3rd way

Advantages

- This form of import allows you to directly access the contents of the addition module as if they were defined in the current module's namespace.
- You can use the functions and variables defined in the addition module without specifying the module name when calling them.
- You can directly use **`add(5, 3)`** without prefixing it with addition.

How _ _init_ _.py file can be used?

In Python packages, the **__init__.py** file can be used for package-level initialization code and to define what gets imported when you import the package as a whole.

Right now, your **__init__.py** is empty. You can add code to it to make your package functions available at the top level, which is a very common and professional practice.

How _ _init_ _.py file can be used?

```
my_calculator/
    __init__.py
    addition.py
    subtraction.py
```

Let's assume you want to make the addition and subtraction functions available directly when you import the my_calculator package. You can achieve this by adding import statements to the **__init__.py** file.

How _ _init_ _.py file can be used?

Edit **__init__.py** file as follows.

```
# my_calculator/__init__.py

# Import functions from addition.py and subtraction.py

from .addition import add

from .subtraction import subtract

print("my_calculator package has been initialized!") # to verify
```

How _ _init_ _.py file can be used?

In this code:

- **‘from .addition import add’** imports the **‘add’** function from the **‘addition’** module within the package.
- **‘from .subtraction import subtract’** imports the **‘subtract’** function from the **‘subtraction’** module within the package.

How _ _init_ _.py file can be used?

Using the package: ***main.py***

Now, when you import the my_calculator package, the **add and subtract functions will be available directly from the package**. You can create a Python script like this:

How _ _init_ _.py file can be used?

Using the package: ***main.py***

```
# main.py

from my_calculator import add, subtract

result1 = add(5, 3)

result2 = subtract(10, 4)

print("Addition result:", result1)

print("Subtraction result:", result2)
```

How _ _init_ _.py file can be used?

Using the package: ***main.py***

Or you can even do it as follows, but then you have to use the package name before the function name every time.

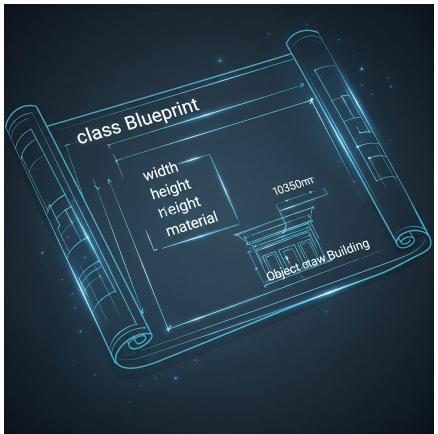
```
import my_calculator

result1 = my_calculator.add(5, 3)

result2 = my_calculator.subtract(10, 4)

print("Addition result:", result1)

print("Subtraction result:", result2)
```

Object-Oriented Programming

What is a Class?

A class is a user-defined data type that defines **a blueprint for creating objects** of that type.

It serves as a template that specifies the **attributes (variables)** and **methods (functions)** that objects created from the class will have.



What is a Class?

```
class Parrot:

    # class attribute

    name = ""

    age = 0
```

In the above example, we created a class with the name Parrot with two **attributes: name and age**.

Class Naming Convention

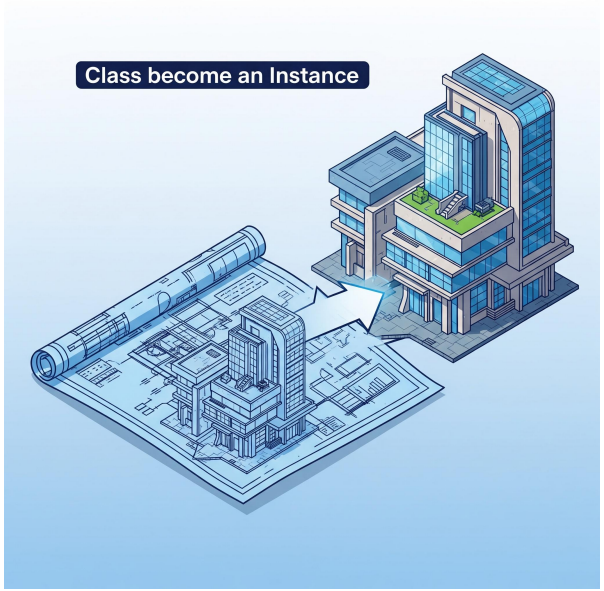
The convention for naming classes is called **CapWords** or **PascalCase** (as per the PEP 8 guideline). It means you **should start every word in the name with a capital letter, without using underscores**.

✅ **Good (Follows Convention):** `MyCar`, `ElectricVehicle`, `User`

❌ **Bad (Works, but Not Recommended):** `myCar`, `electric_vehicle`, `user`

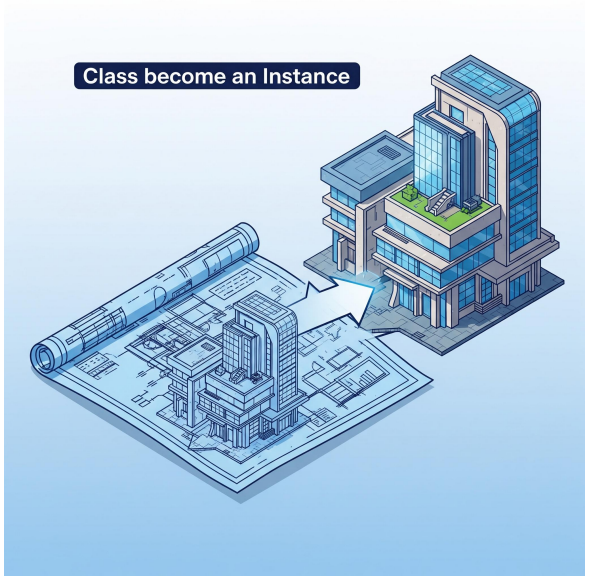
What is an Object?

An object is an instance of a class. In simple words, an **object** is a specific, individual item created from a blueprint.



What is an Object?

Objects can be created based on the class, and they can have their **own unique data (attribute values) while sharing the methods defined in the class.**



Creating an Object?

```
parrot_1 = Parrot()
```

This will create an object named parrot_1 of the class Parrot defined above.

Before diving deep into objects and classes let us understand some basic keywords that will we used while working with objects and classes.

The Python ‘self’

The ‘**self**’ word is used in Python classes to **refer to the instance of the class** on which a method is being called.

It’s **how an object refers to itself within its own methods**.

The Python ‘self’

When you call a method on an object, **Python automatically passes that specific object as the first argument**, which we conventionally name **self**.

When you define methods within a class, you typically **include self as the first parameter for those methods**, **even though you can name it differently if you prefer**.

The Python ‘self’

```python
class Car:
    # CLASS ATTRIBUTE: Shared by all Car objects.
    wheels = 4

    # below are instance methods
    def drive_to(self, destination):
        """This method now requires an additional parameter: 'destination'"""
        print(f"Driving to {destination} on {self.wheels} wheels.")

    def show_wheel_count(self):
        """This method accesses the shared class attribute."""
        # 'self.wheels' correctly finds the class attribute 'wheels'.
        print(f"This car runs on {self.wheels} wheels.")
```

The Python ‘self’

```
---
# 1. Create the instance as usual.
generic_car = Car()

# 2. Call the method and provide the required argument.
# We pass the string "the office" to the 'destination' parameter.
generic_car.drive_to("the office") #Output: Driving to the office on 4 wheels.

# 3. The other method still works without any arguments.
generic_car.show_wheel_count() #Output: This car runs on 4 wheels.
```

The Python _init_() method

In Python, the **__init__()** method (with double underscores before and after "init") is **a special method** or constructor method used **to initialize and customize the attributes of an object when an instance of a class is created**.

It is automatically called when you create a new object from a class.

The Python __init__() method

```
class Car:
    # CLASS ATTRIBUTE: Shared by all Car objects.
    wheels = 4

    # The __init__ constructor is now added.
    # It runs automatically when we create a new Car object.
    def __init__(self, brand, color):
        print(f"A new {color} {brand} car is being created!")
        # INSTANCE ATTRIBUTES: Unique to each car object.
        self.brand = brand
        self.color = color

    def drive_to(self, destination):
        print(f"The {self.color} {self.brand} is driving to {destination}.")

    def show_wheel_count(self):
        print(f"The {self.color} {self.brand} runs on {self.wheels} wheels.")
```

The Python __init__() method

```
# Create two different, unique car objects
car_1 = Car("Toyota", "Blue")
car_2 = Car("Honda", "Red")

print("\n--- Car 1 Actions ---")
car_1.drive_to("the supermarket")
car_1.show_wheel_count()

print("\n--- Car 2 Actions ---")
car_2.drive_to("the beach")
car_2.show_wheel_count()
```

Why ‘self’ is important?

Accessing Object Attributes and Methods:

Inside an **instance method**, **self** is used to access the attributes and methods of the instance.

For example, if you have an attribute **self.name**, you can access it within an instance method as **self.name**.

Similarly, you can call another instance method using **self.another_method()**.

Why ‘self’ is important?

Distinguishing Instance Variables:

self helps **distinguish** between **instance variables** (attributes) **and local variables within a method**.

Why ‘self’ is important?

— — —

Distinguishing Instance Variables:

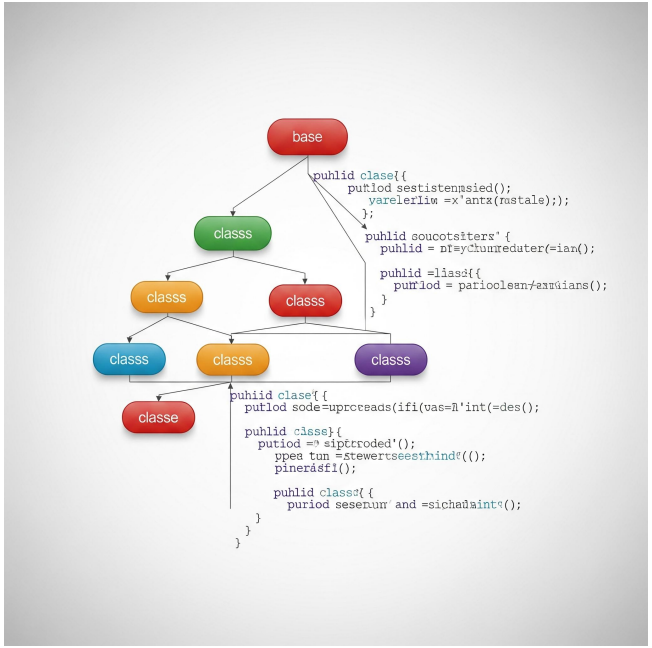
```
class Player:

    def __init__(self, name):
        # 1. INSTANCE ATTRIBUTE: This 'score' belongs to the object.
        # 'self.score' means "this player's own score".
        self.score = 100
        self.name = name
        print(f"{self.name} is starting with a score of {self.score}.")

    def add_points(self, score):
        # 2. LOCAL VARIABLE: This 'score' is just the parameter.
        # It only exists inside this method.
        print(f"--- Adding {score} points to the main score ---")

        # 3. DISTINCTION: 'self' tells Python which is which.
        # This line reads: "My main score now equals my main score plus the new points."
        self.score = self.score + score
```

Inheritance in Python

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows you to create a new class (called **a subclass or derived class**) by **inheriting** properties and behaviors (**attributes and methods**) from an existing class (called **a superclass or base class**).

Inheritance in Python

This is done to **promote code reuse** and build **a hierarchy of classes**.



Inheritance in Python

Base Class (Superclass)

- The base class, also known as the superclass, is the class that **contains the attributes and methods you want to reuse** in other classes.
- It serves as a blueprint for creating derived classes.

Inheritance in Python

```
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        pass  # Placeholder method, to be overridden by subclasses
```

Inheritance in Python

Derived Class (Subclass)

- A derived class, also known as the subclass, is a new class that **inherits attributes and methods from the base class**.
- You **can add additional attributes and methods or override existing ones** in the subclass.

Inheritance in Python

```
class Dog(Animal):
    def speak(self):
        return f"{self.name} says Woof!"

class Cat(Animal):
    def speak(self):
        return f"{self.name} says Meow!"
```

Inheritance in Python

Inheriting & Extending

- In the example above, Dog and Cat are derived classes that **inherit the name attribute and the speak method** from the Animal base class.

- The **speak method in each subclass is overridden** to provide a specific implementation.

Inheritance in Python

Using inherited features

You can create instances of the derived classes and use the inherited attributes and methods.

Inheritance in Python

Using inherited features

```
dog = Dog("Buddy")
cat = Cat("Whiskers")

print(dog.name)      # Output: Buddy
print(dog.speak())   # Output: Buddy says Woof!

print(cat.name)      # Output: Whiskers
print(cat.speak())   # Output: Whiskers says Meow!
```

Inheritance in Python

Example 2:

```
class Person:
    # __init__ is known as the constructor
    def __init__(self, name, id_number):
        self.name = name
        self.id_number = id_number

    def display(self):
        print(self.name)
        print(self.id_number)

    def details(self):
        print("My name is", self.name)
        print("Id Number:", self.id_number)
```

Inheritance in Python

Example 2:

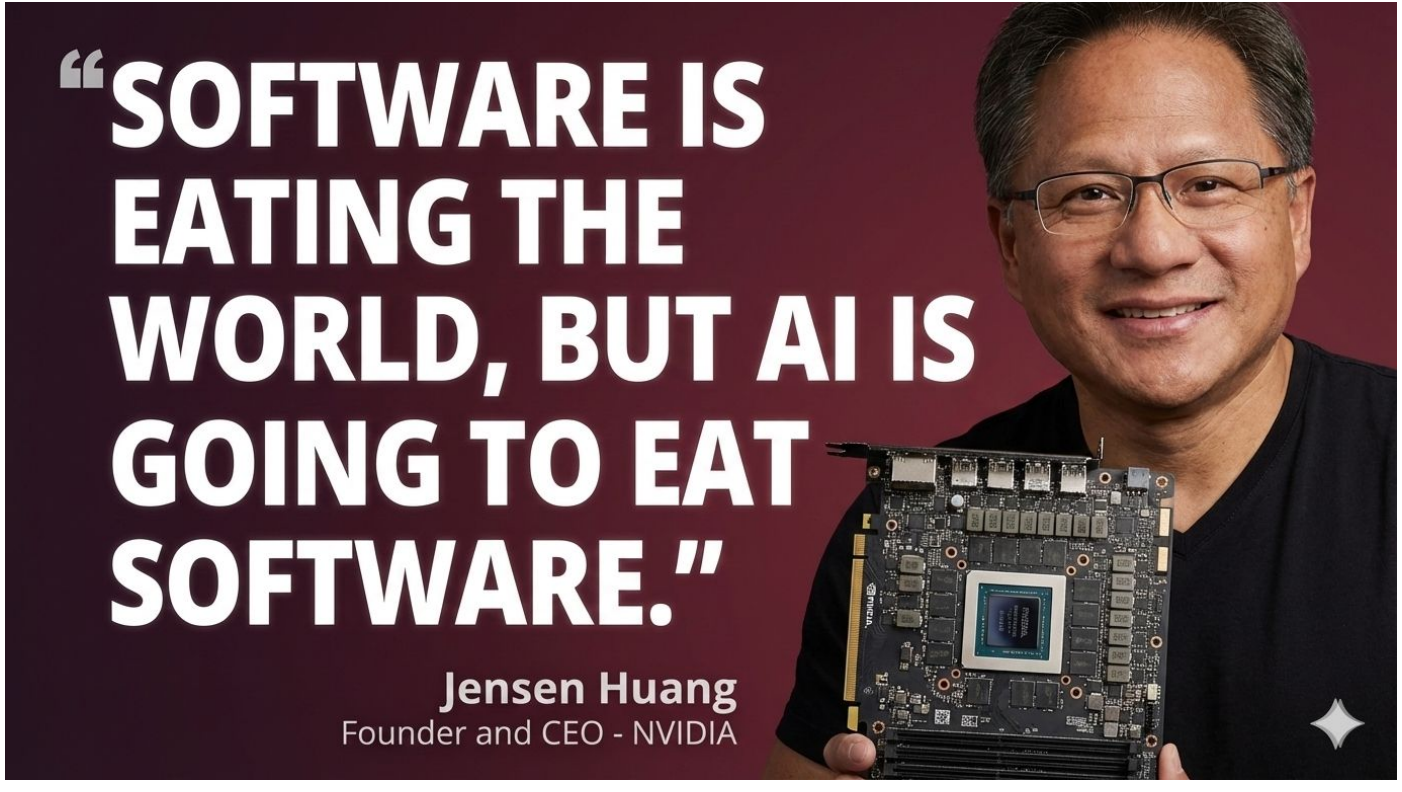
```
class Employee(Person):
    def __init__(self, name, id_number, salary, post):
        self.salary = salary
        self.post = post

        # invoking the __init__ of the parent class
        super().__init__(self, name, id_number)

    def details(self):
        print("My name is ", self.name)
        print("Id Number, ", self.id_number)
        print("Post: ", self.post)
```

Inheritance in Python

Example 2:

```
#Creation of an object variable or an instance
a = Employee('John', 90234509, 200000, "Engineer")

# calling a function of the class Person using
# its instance
a.display()
a.details()
```

Thank You...!


“SOFTWARE IS EATING THE WORLD, BUT AI IS GOING TO EAT SOFTWARE.”
Jensen Huang
Founder and CEO - NVIDIA